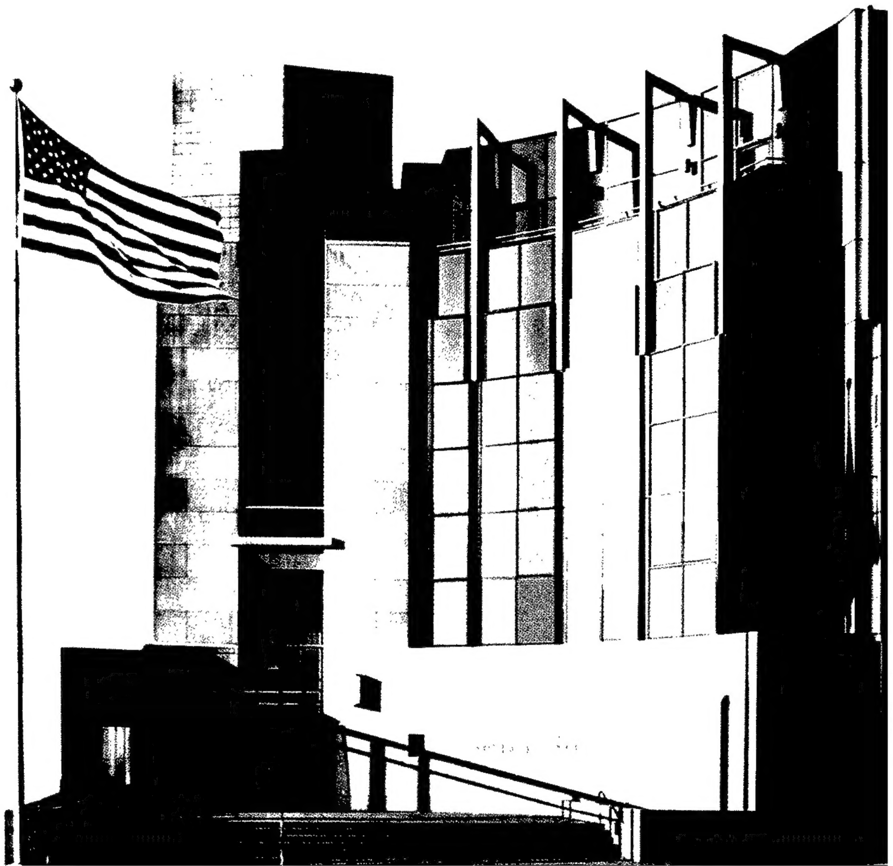
Carnegie Mellon
Software Engineering Institute

rlogin(1): The Untold Story

Lawrence R. Rogers
November 1998



TECHNICAL REPORT
CMU/SEI-98-TR-017
ESC-TR-98-017

This work was funded by AFRL/IFGB.

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

Mario Moya

Mario Moya, Maj, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was funded by AFRL/IFGB. Mr. Peter J. Radesi is the Laboratory Program Manager and he can be contacted at (315) 330-1466.

Table of Contents

List of Figures

Abstract

Coding defects account for a significant portion of the reports received by the CERT® Coordination Center (CERT/CC). Through in-depth analysis of these reports and generalizing our findings from those analyses, we have begun to create guidelines for mitigation strategies for existing defects and avoidance strategies when coding new software. In this document, we report the results of our analysis of the well-known defect in the **rlogin** program. We discuss the coding defect in detail, three mitigation strategies devised to remedy the defect, and two avoidance strategies offered as a guide to reducing the instances of similar coding defects in new programs. We end with three design notes aimed at eliminating these defects at the hardware and protocol design level.

® CERT is registered in the U.S. Patent and Trademark Office.

1 Introduction

We at the CERT Coordination Center (CERT/CC) have analyzed coding defects with the goal of understanding each well enough to communicate the details to those responsible for fixing them (vendors) and those responsible for installing their fixes (systems administrators). Through CERT Advisories, we have met these goals for many years.

Alas, after seeing the same defects over and over, we concluded that this surface analysis was insufficient. We decided that we had to dig deeper to identify the root cause of each coding defect. Once the cause is identified, we can devise mitigation strategies for field repair and avoidance strategies for development environments. We will share our results with the developer and bug-fixing communities to attempt to change the way that program code is written and fixed.

This report describes everything that we at the CERT/CC have learned and subsequently synthesized from analyzing the **rlogin** defect [CERT 97a]. It is intended to stimulate thinking about secure programming practices as well as the design of more secure hardware, operating systems, and protocols.

The discussion begins with an extended description of what **rlogin** is trying to accomplish and then moves to the actual coding defect (Section 2). Next, we examine three mitigation strategies (Section 3), followed by two avoidance strategies (Section 4). Finally, we describe two design notes, one dealing with the **rlogin** protocol and the other with hardware architectures (Section 5).

2 The rlogin Program

2.1 Description of rlogin

Many UNIX systems provide the **rlogin** program. **rlogin** establishes a remote login session from its user's terminal to a remote host computer. Here is an excerpt from Request for Comment (RFC) 1282 that describes the elemental functionality of **rlogin**:

> *The rlogin facility provides a remote-echoed, locally flow-controlled virtual terminal with proper flushing of output. It is widely used between UNIX hosts because it provides transport of more of the UNIX terminal environment semantics than does the Telnet protocol, and because on many UNIX hosts it can be configured not to require user entry of passwords when connections originate from trusted hosts* [Kantor 91].

One feature of **rlogin** is that it passes the terminal type description from the local host computer to the remote host computer. This functionality allows terminal-aware programs such as full-screen text editors to operate properly across the computer-to-computer connection created with **rlogin**.

To do this, **rlogin** passes the user's current terminal definition as identified by the `TERM` environment variable to the remote host computer. RFC 1282 describes how this terminal information is passed from the local host computer, where the **rlogin** client program is running, to the remote host computer, where service is sought.

2.2 Coding Defect in rlogin

Many implementations of the **rlogin** program contain a coding defect where the value of the `TERM` environment variable is copied without due care to an internal buffer. This means that the buffer holding the copied value of `TERM` can be overflowed. On some computer systems, the buffer is a variable local to the `main` subroutine, meaning that the local host computer's subroutine linkage information can be overwritten with data from the `TERM` environment variable.[1] Once overwritten, control can be transferred to an arbitrary address in a computer system's memory.

[1] We now know that buffer overflows that are not stack-based can be just as bad as their stack-based counterparts. See http://www.geek-girl.com/bugtraq/1997_2/0274.html for details.

Overrunning a local variable on the subroutine call stack is called *stack smashing* [Smith 97]. If the data that smashes the stack are carefully selected, control can be transferred to that data. These data are then interpreted as instructions that are subsequently executed by the local host computer.[2] The nature of this code is completely under the control of the **rlogin** program user.

In addition, **rlogin** requires set-user-id root privileges so it can obtain a port in the required range, as described in the **in.rlogind** manual page. Here is an excerpt from that page:

> *The server checks the client's source port. If the port is not in the range 0-1023, the server aborts the connection* [Sun 97a].

More specifically, Figure 1 shows the fragment from **rlogin** that contains the defective code. In this code, data controlled by the user – the `TERM` environment variable – are copied with `strcpy` into the stack-based variable, `term`. If those data in `term` contain instructions appropriate for the local host computer that are properly synchronized with the subroutine linkage requirements, then when `strcpy` returns to its caller, it will instead return to the code that just smashed the stack. To compound the problem, **rlogin** has not yet shed its root privileges at the time that the `strcpy` subroutine is called. That happens with the `setuid` call later in the code. This means that when the smashed stack instructions are executed, they run with full root privileges.

```
main(argc, argv)
  int argc;
  char *argv[];
{
        ...
        int uid;
        long omask;
        struct passwd *pw;
        char *host, *p, *user, term[1024];
        struct servent *sp;

        ...
        if (!(pw = getpwuid(uid = getuid()))) {
                (void)fprintf(stderr, "rlogin: unknown user id.\n");
                exit(1)
        }
        ...
        (void)strcpy(term, (p = getenv("TERM")) ? p : "network");
        ...
        rem = rcmd(&host, sp->s_port, pw->pw_name, user, term, 0);
        ...
        (void)setuid(uid);
        doit(omask);
        ...
}
```

Figure 1: Defective Code Segment

[2] There are many references that describe how to select these data: [One 96, Mudge 96, Mudge 95, Lefty 96, Prym 96].

To exploit this coding defect, one need only craft the appropriate value for the `TERM` environment variable, place it in the environment that **rlogin** will inherit, and then run **rlogin**. In the exploitation scripts that we have seen at the CERT/CC, the code that smashes the stack starts a copy of `/bin/sh`, one of several standard command language interpreters typically found on a UNIX system. The user can then execute any commands that he or she chooses, and execute them as root.

2.3 Determining Vulnerability

For a computer system to be at risk, the computer system's hardware and software architecture must support a program's ability to determine the location of the subroutine linkage information. Further, once located, that architecture must support a program's ability to change that information so that execution can continue at an arbitrary location in memory.

On most modern computer hardware and software architectures, a subroutine's local variables are intermixed with subroutine linkage information. This means that the location of the linkage information can be computed given the addresses of local variables. In fact, several architectures place linkage information adjacent to a subroutine's local variables. Because of this adjacency, exceeding the size of the storage allocated to one of the subroutine's local variables can also change the subroutine linkage information. The key points here are the ability to determine the location of the linkage information and to change it. The adjacency attribute simplifies the location determination step. It is an aid, not a necessity.

The ability to execute instructions located in an arbitrary portion of a computer system's memory is another key hardware and software architectural requirement. Some hardware architectures, notably Sun Microsystems' SPARC® architecture, support the operating system's ability to define a section of memory as being non-executable. This means that on a SPARC-based machine, the operating system can mark the instruction segments as executable and all other segments as non-executable. Execution control can be successfully transferred only to instruction segments, not an arbitrary location in memory. On the other hand, the Intel Pentium® hardware architecture does not have the ability to enforce these restrictions. Control can be transferred to an arbitrary location in memory where execution can continue.

3 Mitigation Strategies

What options do system managers have to reduce the risks to their systems once a defective version of **rlogin** is installed? This section describes three mitigation strategies. They are presented in order from least effective to most effective.

3.1 Strategy 1: Non-Executable Stack Regions

The first mitigation strategy consists of removing execution permission from the stack segment of every process on a UNIX system. This means that the class of exploit scripts that smash the stack and then execute instructions placed on the stack will not work. Methods to remove execution permission from the stack are described in [Sun 97b] and [Dik 97]. This strategy makes no changes to the **rlogin** source code and as such does not require that code to proceed.

Unfortunately, some UNIX systems make legitimate use of an executable stack. For example, Linux uses executable stacks for signal handling trampolines. Objective C uses other types of trampolines that also require an executable stack. There are likely others.

To that end, then, is removing stack execution permissions worth the effort required to fix the problems that it creates? Let's look further at what it means to make such a change.

First, removing stack execution permissions only renders data on the stack non-executable. Second, not all processor memory management units provide the granularity necessary to enforce these permissions. Finally, even if so protected on computers that support it, the subroutine call stack can still be corrupted. This means that when the subroutine whose stack has been corrupted attempts to return to its caller, that corrupted stack could transfer control to a non-stack-based address, an address in the environment variables section for example. Execution can continue from that point.

While the exploit scripts being used today will no longer work, we believe that those scripts can be easily changed so that they work for a system with a non-executable stack. The exploit scripts can still cause a buffer overflow that corrupts the subroutine call stack and ultimately executes arbitrary code located somewhere other than on the stack. Therefore, while this mitigation strategy is a minor improvement, it is incomplete and has a potentially high cost.

3.2 Strategy 2: Truncating the Data

The next mitigation strategy suggests that the data copied from the TERM variable be truncated to fit into the term buffer. Some data may be lost if the size of the TERM variable exceeds that of the term buffer. How the remote computer system reacts to the truncated terminal name is unpredictable.

There are two ways to implement this strategy. The first requires changing the code for **rlogin**. It is easy to apply: Simply replace the instance of strcpy with strncpy followed by inserting the NULL terminator at the end of the term string. Figure 2 shows the resulting code.

```
(void)strncpy(term, (p = getenv("TERM")) ?
                  p : "network", sizeof(term));
term[sizeof(term) - 1] = (char) NULL;
```

Figure 2: Truncating the Data

Many vendors used this method in the patches provided to their customers. It avoids the buffer overflow, and it is easy to understand and apply.

The second implementation method consists of replacing **rlogin** with a wrapper program that truncates the TERM variable before **rlogin** can operate on it. This strategy does not rely on the source code. This scheme was proposed in [CERT 97a].

This strategy is an improvement. There are neither buffer overflows nor smashed stacks, but the data resulting from the truncation operation may yield unpredictable results.

In addition, the replacement code does not check content. This means that potentially harmful data are passed from computer to computer, perhaps causing problems along the way. The next section suggests a way to solve this problem.

3.3 Strategy 3: Data for Length and Content

The key to this mitigation strategy is to inspect both the TERM variable's length and its content. Data that do not fit or are incorrectly formed should be replaced by a meaningful default value to ensure predictable results further down the line. As in Strategy 2, a wrapper or changes to the source code can also implement this strategy. We will discuss the suggested changes to the source code to achieve our desired result.

To begin, there is nothing fundamentally wrong with the `strcpy` subroutine as long as the data to be copied fit in the destination area provided. To that end, one need simply check the length of `TERM` to see if it fits in the destination. If it does not fit, a suitable replacement should be copied instead.

Next, let's consider the content of `TERM`. We ask: What is the correct definition of a terminal name object? All too often in the C programming language, the universal answer is "It's a string" or "It's an integer." In this case, the fundamental data type is not just a string, but rather a string with a length and content definition.

Sadly, there are no standards or RFCs that define the form of a terminal name object. Instead, we will have to deduce those attributes through ad hoc techniques. To do this, we will examine a Sun Microsystems Solaris 2.5.1 system to help us define the form of a terminal name object.

By looking at all of the terminal names used by the termcap and terminfo libraries in Solaris 2.5.1, we observe that the maximum size of a terminal name object is 26 characters. To be safe, we will select 64 characters as the maximum. Further, we observe that the character set appears to be drawn from the alphanumeric set plus the special characters plus (+), minus (–), period (`.`), and forward slash (`/`). We will use that character set to define valid content.

Figure 3 shows the improved code segment based on the defective code in Figure 1. The improved code uses this newly defined terminal name object. This code segment recognizes that data that cross a boundary – the boundary between user and program – need to be inspected before the data are used. (See Section 4.1 for a description of boundaries.)

The code inspects the data by checking the length and content of the value of the user-provided `TERM` variable before sending it to the remote host computer. The `strcpy` subroutine can safely operate without causing a buffer overflow that would render the local host system vulnerable to attack. The wrapper that is proposed in Section 3.2 could do the same thing shown in Figure 3. This mitigation strategy is the most complete of the three presented.

```
#define DEFAULT_TERM          "network"
#define MAX_TERM_LENGTH       64

static char *Term_OK_Chars = "0123456789abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ+-/.";

main(argc, argv)
  int argc;
  char *argv[];
{
        ...
        int uid;
        long omask;
        struct passwd *pw;
        char *host, *p, *user, term[MAX_TERM_LENGTH];
        struct servent *sp;

        ...
        if (!(pw = getpwuid(uid = getuid()))) {
                (void)fprintf(stderr, "rlogin: unknown user id.\n");
                exit(1)
        }
        ...
        if ( ((p = getenv("TERM")) == (char) NULL)    ||
             (strlen(p) >= sizeof(term))              ||
             (strspn(p, Term_OK_Chars) != strlen(p)))  {
                p = DEFAULT_TERM;
        }
        (void)strcpy(term, p);
        ...
        rem = rcmd(&host, sp->s_port, pw->pw_name, user, term, 0);
        ...
        (void)setuid(uid);
        doit(omask);
        ...
}
```

Figure 3: Checking for Length and Content

3.4 Summary and Implementation

This section summarizes the mitigation strategies just discussed. It also proposes an alternative implementation using wrapper technology.

3.4.1 Summary of Mitigation Strategies

The first strategy, which was explained in Section 3.1 and which made **rlogin**'s stack not executable, is effective against the current exploit scripts. However, we feel that those scripts need only minor changes to work correctly again. Therefore, that strategy is not recommended as a complete solution to the coding defect in **rlogin** and similar defects in other programs. Nonetheless, where possible and practical, stack regions should be made not executable.

The second strategy, explained in Section 3.2, suggested truncating the `TERM` variable to a defined length at the cost of unpredictable results. This strategy could be achieved through a wrapper program or changes to **rlogin**'s source code. Because this strategy can produce unpredictable results, it too is not recommended, though it has been widely used by vendors.

The last strategy, explained in Section 3.3, suggested that TERM's value be checked for length and content before being used. The strategy also suggested that a reasonable default be substituted in the absence of conforming data. A wrapper or changes to source code are two implementation methods. This strategy is recommended because it does mitigate the current exploits and ensures predictable results.

3.4.2 Wrapper Technology

The last two strategies suggested a wrapper as a method for rendering the current exploit scripts inoperative. Wrappers do indeed work, but their focus is narrow. A wrapper addresses only **rlogin**'s defect and nothing else. Similar defects in other programs require their own wrappers.

Consider then a universal table-driven wrapper as a general-purpose solution. This wrapper would intercede between the user and the program actually providing service in much the same way that the TCP Wrappers [Venema 97] stand between a client program requesting a network service using the Transmission Control Protocol (TCP) and the daemon that provides it.

The wrapper could be directed not only to check environment variables for form and content but also to validate arguments, reset resource limits, alter the disposition of signals, close unnecessary file descriptors, and reestablish credentials. In short, it would examine all process attributes that are preserved across the exec(2) family of UNIX system calls. The wrapper could be directed to set the state of a soon-to-be-executed program to a known and well-defined state based on a table that describes the operations it is to perform.

To install this general wrapper on a UNIX system, every set-user-id and set-group-id program must first be relocated to another place in the file system. Once relocated, the set-user-id and set-group-id permissions must be removed. This ensures that even if these defective programs were executed directly – that is without benefit of the wrapper – they would not yield additional privileges to the executing user. The wrapper would then be installed in place of those relocated set-user-id and set-group-id programs with the original permissions restored. Lastly, the administrator needs to construct the table that defines the wrapper's operations.

Not all sources of data are passed from process to process through the exec system calls. Programs also access data streams using file and network descriptors created as a program executes. Data in those streams should also be cleansed where possible.[3] The general wrapper described here does not operate on that data stream, even in the specific case for a known application and therefore a known data stream form. In the general case, the cleansing task is even harder.

[3] See the defect in **rpc.statd** (http://www.cert.org/advisories/CA-97.26.statd.html).

The wrapper implementation method does reduce the effect of the type of coding defect found in **rlogin**. However, the language needed to describe how to cleanse arguments, environment variables, etc., may be cumbersome and therefore error prone. It seems likely that legitimate combinations of arguments, environment variables, and such may be unnecessarily cleansed, perhaps at the cost of desired functionality. Finally, the wrapper is not able to cleanse all sources of data available to an arbitrary program. While wrapper technology is more practical when the source is unavailable, its lack of coverage and complexity are drawbacks. Therefore, wrappers should be used only when there are no other choices.

In summary, the best strategy suggested checking a data item for content and length, and the best implementation method suggested changing the source code for the program in need of repair. Because the best strategy required source code, it is likely a strong avoidance strategy candidate as well. The next section describes general concepts for avoiding common coding defects of which **rlogin**'s buffer overflow is but one type.

4 Avoidance Strategies

The purpose of avoidance strategies is to eliminate the need for mitigation strategies. Programs written in anticipation of extraordinary conditions are more able to operate correctly than programs written without such foresight. In other words, programs written correctly from the beginning need not have coding defects repaired later in their life cycle.

Programs will evolve. Their requirements will change and their functionality will likely increase to meet this demand. However, the implementation of that functionality should be achieved without the programming flaws that frequently characterize modern software.

At the CERT/CC, we have coined the phrase *defensive programming practices* to encapsulate this concept. It is patterned after the defensive driving techniques that many of us were taught when we took Driver's Education in secondary school.

Driver's Education teaches prospective drivers to be prepared for unexpected driving situations. Examples are sudden stops, lane changes, road hazards, and mechanical malfunctions. A defensive driver will be in a position to safely navigate the highway in spite of these conditions.

The analog for programming is simple: Assume that those using your programs will not just provide bad input to your program as the result of their ignorance; they will also consciously provide malicious input designed to make your program operate in an unintended fashion. The term *input* includes not only the files that a program reads and writes but also the entire scope of a program's operating conditions. Examples are arguments, environment variables, credentials, open file descriptors, data streams, and system resources. A defensively written program anticipates anomalies in such inputs.

For **rlogin**, the user did not just accidentally provide a TERM environment variable that happened to contain machine language instructions properly aligned with the requirements of the subroutine call stack. He or she consciously created such a variable for the expressed purpose of gaining access to resources to which that user was not entitled. The mitigation strategy that inspected the data for length and content becomes the avoidance strategy that renders such unwanted access impossible.

This section discusses two examples of defensive programming practices. The first deals with examining data for length and content, and the second with program privileges.

4.1 Practice 1: Trusting Untrustworthy Data

Sections 3.2 and 3.3 pointed out that data under the control of the user should be examined before being used. This section covers the specific case of data that crosses a *boundary* – an imaginary line separating two potentially competing domains. These data must be considered *untrustworthy* and be made *trustworthy* before being used. The nature of the domains on either side of these boundaries is the subject of the discussion below. Indeed, their very existence is a new concept about which programmers should be aware.

A boundary typically separates two potentially competing domains. The fact that these domains compete, or rather have competing goals, frequently gives rise to attacks from computers on one side of a boundary targeted towards computers on the other side. Examples are attacks against the Domain Name Service (DNS) [CERT 97b] and the Internet Protocol (IP) [CERT 96]. Firewalls and other perimeter defenses are traditional countermeasures used to stop these attacks. The firewall defines a boundary crossed by data that are untrustworthy.

In **rlogin**, there is a boundary between the program – more accurately, the programmer who designed and wrote the program – and its user. When `TERM` crossed this boundary, the programmer, and subsequently the program, did not recognize that this boundary existed and did not code appropriately. Had the programmer recognized this, the mitigation strategy described in Section 3.3 could have been used to make the untrustworthy data trustworthy.

When the **rlogin** user is intending to be malicious, it is also clear that the programmer and the user have competing goals. The programmer's goal is to provide virtual terminal service as defined by the relevant RFCs. In contrast, the malicious user's goal is to gain extraordinary access to the computer system where a defective **rlogin** has been installed. Because of the coding defect, the malicious user's goals can be achieved.

In the specific case of **rlogin**, there are other places where data cross a boundary. Two boundaries are listed below. In all cases, the boundary exists between the programmer or program and the domain listed.

4.1.1 Boundary 1: The Program User

This boundary has been discussed with respect to environment variables, but there is at least one other boundary, namely program arguments. **rlogin** expects the following arguments:

```
[-e char] [-l username] host
```

where

- `char` is the escape character that when typed allows the user to control **rlogin**'s actions. The value must be a legal ASCII character. This argument is optional.

- `username` allows the user to specify a different user name for the remote host computer login. If this option is not used, the local user name is used. The value must adhere to the rules used to name users on the remote host computer, likely a string composed of eight or fewer alphanumeric characters drawn from the ASCII character set. This string must be able to be mapped onto an account on the remote host computer, usually by an entry in the *`/etc/passwd`* file. This argument is optional.
- `host` is the name of the remote host computer from which virtual terminal service is sought. The value here must conform to the host naming standards defined in RFC 952 [Harrenstien 85] and 1123 [Braden 89].

All arguments should be inspected for content and length before they are used.

4.1.2 Boundary 2: Administrative Domains

Another boundary is the boundary between administrative domains. In this case, the domains that we are concerned about are those that are mapping host names to IP addresses and vice versa. Usually, this mapping is done by data provided by a DNS server. In many cases, these servers are beyond the administrative control of the **rlogin** user. The data exchanged must be inspected for form and content as before, but then one additional check should be made.

If DNS servers are set up properly, the primary or canonical name of a host should be able to be mapped to an IP address and then that address mapped back to a host name. The primary host name should be the same as that host name discovered by the twice-applied mapping process, independent of upper vs. lower case issues. If the host names are not the same, then the administrator of the remote DNS server may be attempting to gain unauthorized access to a resource, likely to a host to which they are not entitled such access.[4] It is then reasonable to assume that something is amiss and that attempting a virtual terminal connection should not proceed. **rlogin** should first do this double mapping, and then continue only if valid data were found. If an invalid mapping is discovered, **rlogin** should describe its findings and then abandon the attempt to connect to the specified remote host.

4.1.3 The Practice

Making untrustworthy data trustworthy consists of the following steps:

1. Identify the boundaries in a program.
2. Identify data that cross boundaries in that program.
3. Examine that data for correct form, substituting meaningful and predictable defaults for nonconforming data.

[4] This is not the only reasonable conclusion to draw. DNS servers that do not provide consistent information across the twice-mapped method may simply be badly managed by administrators who do not know how to configure them correctly.

4.2 Practice 2: Shedding Privileges

From a previous discussion, we know that **rlogin** must run with root privileges to gain access to a reserved port as part of the authentication process. While this is a poor authentication scheme (see Section 5.1), the **rlogin** program must nonetheless be securely programmed to implement that scheme. To that end, where specifically does **rlogin** need root privileges? The goal we are trying to achieve here is to use extraordinary privileges only where needed and then give them up when they are no longer required. This section describes our attempts to achieve this goal by several different methods.

rlogin needs root privileges so that the `rcmd` subroutine – the subroutine that makes the connection to the remote host computer – can succeed. Here is an excerpt from the `rcmd` manual page:

> *rcmd() is a routine used by the super-user to execute a command on a remote machine using an authentication scheme based on reserved port numbers. rresvport() is a routine which returns a descriptor to a socket with an address in the privileged port space* [Sun 97c].

4.2.1 Temporarily Abandoning Privileges

All of the code up to the `rcmd` call as shown in Figure 3 – including the defective code that uses the `TERM` environment variable – does not need root privileges to operate, and all of the code after `rcmd` does not need privileges either. If privileges can be temporarily given up, reclaimed for `rcmd`, and then given up completely, **rlogin** can operate more safely with privileges being enabled only when needed.

On a Solaris 2.5.1 system, this temporary privilege reduction and reclamation task can be accomplished because of IEEE Standard Portable Operating System Interface for Computer Environments (POSIX) saved set ids [POSIX 98] and the `seteuid` system call [Sun 97d]. Figure 4 shows how the original privileges are saved for later use with `getuid` and `geteuid`, the current privileges are reduced and reclaimed with `seteuid`, and then privileges are irrevocably abandoned with `setuid`.

```
#define DEFAULT_TERM          "network"
#define MAX_TERM_LENGTH       64

static char *Term_OK_Chars = "0123456789abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ+-/.";

main(argc, argv)
  int argc;
  char *argv[];
{
        ...
        int original_ruid, original_euid;
        long omask;
        struct passwd *pw;
        char *host, *p, *user, term[MAX_TERM_LENGTH];
        struct servent *sp;

        original_euid = geteuid();
        seteuid(original_ruid = getuid());
        ...
        if (!(pw = getpwuid(original_ruid))) {
                (void)fprintf(stderr, "rlogin: unknown user id.\n");
                exit(1)
        }
        ...
        if ( ((p = getenv("TERM")) == (char) NULL)    ||
             (strlen(p) > sizeof(term))               ||
             (strspn(p, Term_OK_Chars) != strlen(0)))  {
                p = DEFAULT_TERM;
        }
        (void)strcpy(term, p);
        ...
        seteuid(original_euid);
        rem = rcmd(&host, sp->s_port, pw->pw_name, user, term, 0);
        ...
        (void)setuid(original_ruid);
        doit(omask);
        ...
}
```

Figure 4: Only Use Privileges When Needed

Sometimes, `seteuid` is the right call to use. It provides the flexibility needed to obtain a privilege when required, yet it affords the chance to shed that privilege. Using `seteuid` seems to be an effective strategy and was easy to implement. However, by slightly perturbing the machine language instructions copied to and then executed from the stack, these temporarily waylaid privileges can be reclaimed, rendering the strategy ineffective. Therefore, if privileges are required, they must be used and then abandoned completely. Temporarily casting privileges aside does not work. This means that the code fragment shown in Figure 4 is not a good solution.

4.2.2 Permanently Abandoning Privileges

Within the confines of the way that `rcmd` is currently designed and implemented, the notion of using privileges at the beginning of a program, **rlogin** for example, and then abandoning them is impossible. However, if the reserved port could be allocated independent of `rcmd` and then given to `rcmd` when needed, root privileges could be given up immediately after that port is allocated. The intervening code could run as the real user without a loss of functionality.

From `rcmd`'s manual page description, we know that it uses the `rresvport` subroutine to allocate a privileged port. Unfortunately, there is no other way to give that port to `rcmd`. As a solution to this dilemma, we suggest that a new version of `rcmd` be written that accepts a reserved port given as another argument. In fact, `rcmd` uses two privileged ports if its last argument is non-zero. This improved version must take this into account because programs such as **rsh** use this argument. With this new version, the ports that `rcmd` needs can be given as arguments, thereby removing `rcmd`'s need to run as root.

If we look closely at the source code for `rcmd` as shown in Figure 5, we see that it tries repeatedly to connect to the remote host computer. `rcmd` anticipates some subtle system timing issues whereby a reserved port bound to an imprecise destination may be superceded by that same port bound to a more specific destination. In response to this, `rcmd` allocates another reserved port and attempts the connection again.

With the reserved port allocation code removed from our new version of `rcmd`, `rcmd` will not be able to connect to the remote host computer if the selected port is not able. This is an acceptable change as long as `rcmd` reflects this state to its caller.

For example, **rlogin** could re-execute itself to reclaim root privileges safely. The reserved port allocation procedure would begin again, and if successful, the remote connection would also be retried. Apart from minor performance issues, the **rlogin** user should not see any change in functionality.

With these proposed changes to `rcmd`, it is now up to **rlogin** to allocate the reserved port and then give up its root privileges. **rlogin** can easily allocate that port with `rresvport` but giving up privileges is a little harder and, as we will see, non-portable. We will look first at the Solaris implementation in detail and then talk about other versions of the UNIX Operating System.

There are two ways for a set-user-id Solaris process to irrevocably abandon its privileges. The key to both methods is their effect on the POSIX saved set ids.

The first way uses the `setuid` system call. If the effective id of the process is root, the `setuid` system call works fine. That is, it sets the real, effective, and saved set user ids to its argument, thereby giving up privileges forever. However, if the effective id is not root, then the saved set id value remains unchanged. In this case, privileges are given up only temporarily. As we noted earlier, they can be reclaimed. Therefore, in the case of set-user-id root programs, `setuid` is a complete solution, but in the case of set-user-id-not-root programs, it is not.

```
for (timo = 1, lport = IPPORT_RESERVED - 1;;) {
        s = rresvport(&lport);
        if (s < 0) {
                if (errno == EAGAIN)
                        (void)fprintf(stderr, "rcmd: socket: All ports in use\n");
                else
                        (void)fprintf(stderr, "rcmd: socket: %s\n", strerror(errno));
                sigsetmask(oldmask);
                return (-1);
        }
        fcntl(s, F_SETOWN, pid);
        sin.sin_family = hp->h_addrtype;
        bcopy(hp->h_addr_list[0], &sin.sin_addr, hp->h_length);
        sin.sin_port = rport;
        if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) >= 0)
                break;
        (void)close(s);
        if (errno == EADDRINUSE) {
                lport--;
                continue;
        }
        if (errno == ECONNREFUSED && timo <= 16) {
                (void)sleep(timo);
                timo *= 2;
                continue;
        }
        if (hp->h_addr_list[1] != NULL) {
                int oerrno = errno;

                (void)fprintf(stderr, "connect to address %s: ",
                        inet_ntoa(sin.sin_addr));
                errno = oerrno;
                perror(0);
                hp->h_addr_list++;
                bcopy(hp->h_addr_list[0], &sin.sin_addr, hp->h_length);
                (void)fprintf(stderr, "Trying %s...\n",
                        inet_ntoa(sin.sin_addr));
                continue;
        }
        (void)fprintf(stderr, "%s: %s\n", hp->h_name, strerror(errno));
        sigsetmask(oldmask);
        return (-1);
}
```

Figure 5: rcmd Source Code

The second way is with the `setreuid` system call. The key phrase from the Solaris 2.6 manual page [Sun 97e] is provided below:

> *... if the real user ID is being changed (that is, if **ruid** is not -1),or the effective user ID is being changed to a value not equal to the real user ID, the saved set-user ID is set equal to the new effective user ID .*

This means that by setting the real user id to its current value, that is a value other than –1, and also setting the effective user id to the real user id, the saved set user id is also set to the real user id. All three ids have the same value, namely the value of the real user id. Privileges are then irrevocably abandoned. Figure 6 shows the final version of **rlogin** that uses our new version of `rcmd` and the concept of using and then shedding privileges as soon as possible.

```
#define PATH_RLOGIN            "/usr/bin/rlogin"
#define DEFAULT_TERM           "network"
#define MAX_TERM_LENGTH        64

static char *Term_OK_Chars = "0123456789abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ+-/.";

main(argc, argv)
  int argc;
  char *argv[];
{
        ...
        int reserved_port;
        char *host, *p, *user, term[MAX_TERM_LENGTH];
        struct servent *sp;

        /* Begin critical region - this region operates as root */
        for (reserved_port = IPPORT_RESERVED - 1;;) {
                if (rresvport(&reserved_port) < 0) {
                                /* error handling */
                }
        }
        setreuid(getuid(), getuid());
        /* End critical region - this region operates as the
        ...
        if ( ((p = getenv("TERM")) == (char) NULL)    ||
             (strlen(p) > sizeof(term))              ||
             (strspn(p, Term_OK_Chars) != strlen(0)))  {
                p = DEFAULT_TERM;
        }
        (void)strcpy(term, p);
        ...
        rem = new_rcmd(&host, sp->s_port, pw->pw_name, user, term, 0,
                        reserved_port, 0);
        if (rem == ERROR_CANNOT_BIND_TO_PORT) {
                execv(PATH_RLOGIN, argv);
                perror(argv[0]);
                exit(1);
        }
        ...
        doit(omask);
        ...
}
```

Figure 6: Irrevocably Abandoning Privileges

Other operating systems, Hewlett-Packard's HP-UX for example, contain the `setresuid` [HP 97] system call where the values of the real, effective, and saved set user ids can be set directly. The saved set user id is not set based on a side effect of setting other values. Instead, `setresuid` lets you set these values directly, and their setting is clear from reading the program code.

4.2.3 The Practice

When you need extraordinary privileges to accomplish a programming task, you should do the following:

1. Use privileges at the beginning of your program.
2. Give up privileges immediately after you are done with them.
3. Give up privileges in a manner that they cannot be reclaimed. We know that temporarily giving up privileges only slows the intruder but does not stop the attack.

4. Write concise privileged code. Privileged code should be concise so that its operation can be thoroughly inspected for defects.

5. Change the programming interfaces to vendor-provided software so you can isolate code segments that require privileges.

5 Design Notes

This section details three design notes aimed at eliminating the types of problems that rendered **rlogin** and other similar program vulnerable to attack. They address protocol and hardware design issues.

5.1 Note 1: Weak Authorization

Let's summarize the **rlogin** problem. The **rlogin** protocol specifies that each connection from an **rlogin** client to an **rlogin** server must originate on a privileged port, that is, a port in the range of 512 to 1023 inclusive. On a UNIX system, this means that the client must have root privileges to gain access to these ports. Because of a coding defect, an intruder can use this extraordinary privilege to gain access to resources to which they are not entitled. This entire problem is predicated on the simple idea that using privileged ports conveys a sufficient level of authorization by the client. Let's look more closely at what is really going on here.

The protocol requires that the source port number be within a specific range. Who sets that value? The originating host does, meaning that that host is responsible for an important aspect of the connection. Furthermore, there is no means to verify the authenticity of the source port number. It is meaningless to ask the attacker's operating system if the source port number is correct and is being set properly. The **rlogin** server is, in essence, trusting untrustworthy information that cannot be vetted.

The bottom line is that the **rlogin** protocol as defined in RFC1282 is partially flawed because it relies on information – the source port number – that cannot be verified. The design note to be gleaned from this analysis is that unverifiable information must not be used, and certainly must not be used in a network protocol.

5.2 Note 2: Subroutine Linkage Information

One of the key features of the buffer overflow in **rlogin** is the ability to redirect the flow of execution to a memory location whose contents can be defined by the **rlogin** user. This is straightforward to accomplish given the stack-based architecture of most modern computers. Although recent discussions on the BUGTRAQ mailing list have suggested randomizing the location of the subroutine linkage information [Kettlewell 98], this does not defeat – nor is it intended to defeat – all attack types. This design note recommends a change in the location of the subroutine linkage information and the rules that define how and when that information can be modified under program control.

We propose a change to the architecture of computers such that, at a minimum, the subroutine return addresses be stored in their own protected area of computer memory. To implement this, we recommend the following changes:

- Add one register to the computer's register set. This register is called the return address stack pointer (RASP). Its contents will always point to a memory location so it should be sized appropriately.
- Change the operation of the subroutine call and return instructions to use the RASP. The subroutine call instruction pushes the return address onto the memory cell pointed to by the RASP and then transfers control to the specified address. Similarly, the subroutine return instruction pops the return address from the memory cell pointed to by the RASP and transfers control to that address. These instructions operate just as before except that they use the RASP instead of the general-purpose stack pointer.
- Make RASP and the return address pool read only when the computer is operating in user mode. The only exceptions are the subroutine call and return instructions as noted above. There are no access restrictions when in kernel mode.

With these changes to the architecture, what software changes must be made? The following list highlights the areas that need attention.

- The kernel. The kernel needs to manage this new data structure both on behalf of processes and the kernel itself. The memory area used to hold the pool of return addresses should be allowed to grow as a process evolves, similar to the way that the stack grows as needed.
- Debuggers and postmortem analyzers. These tools need to know about the RASP and the pool of return addresses so that they can properly display the stack trace that shows the order of subroutine calls and the parameters passed to those subroutines.
- All programs that use `setjmp`(3) and `longjmp`(3) [Sun 97f]. The `setjmp` and `longjmp` subroutines, available on most UNIX systems, provide a non-local `goto` to languages that do not have such a facility. Programs can save many key components of process state – usually the hardware register set and the current instruction pointer – with `setjmp` and then return to that state with `longjmp`. While `setjmp` can record the RASP, `longjmp` will be unable to set RASP to the saved value. Programs that use the `setjmp`/`longjmp` pair must be rewritten to use either
  - conventional subroutine returns. This is no small task. For example, in FreeBSD Version 2.2.6, there are over 175 calls to `setjmp` affecting over 50 different programs!
  - a different programming interface to `setjmp`/`longjmp`. We suggest moving the `setjmp`/`longjmp` functionality into the kernel. Instead of retaining state information in a buffer held in the address space of a user process, the kernel would manage the state information. `setjmp` would return a handle to an instance of state, and `longjmp` would restore the state associated with that handle. Programmatically, the changes needed in those 175 calls to `setjmp` noted above would still be needed, but the overall logic of the associated 50 programs would not change.

The changes proposed in this section end the rash of buffer overflows that have been recently brought to light. They provide a hardware solution that requires some complementary software changes. These changes do not require application-level changes like those presented in Sections 3 and 4. We urge computer hardware designers to consider the suggestions put forth here and to implement them where possible.

5.3 Note 3: Memory Management Units

Section 3.1 described a mitigation strategy that advocated removing execution permissions from the stack segment of every process running on a UNIX system. While this strategy has its limitations, it does defeat many of the exploitation scripts presently in use on the Internet.

The strategy also noted that removing such permissions was not always possible; that is, not all memory management units provide the capability to enforce such restrictions at the hardware level. Specifically, while Sun Microsystems' SPARC architecture does provide this capability, Intel's Pentium architecture does not.

We recommend that a memory management unit support all possible combinations of access permissions at the lowest possible level, for example at the page level. By supporting all combinations, operating systems developers can enforce whatever protection model they deem appropriate for their domain. Further, by providing these permissions at the lowest possible level, operating systems developers are encouraged to use these facilities because they are practical and complete.

6 Summary and Future Work

This report describes the results of applying the detailed vulnerability analysis process to the defect in **rlogin**. We first analyzed the code to determine the root cause. We decided that the problem is not just a buffer overflow; rather, it is an instance of trusting untrustworthy data. By recognizing this more encompassing classification, we devised a mitigation strategy (Section 3.3) that fixed the defect in a way that operated predictably. Given the fundamental flaw, we wrote a defensive programming practice (Section 4.1) aimed at eliminating these types of defects in future products.

We could have stopped here; the defect was successfully repaired and we had written the practice describing the conditions necessary to recognize the flaw and the steps to follow to prevent a reoccurrence. We chose to go further to see what else we could find.

Through further analysis, we also discovered that the underlying computer hardware could be changed to prevent related defects. These gave rise to two design notes (Sections 5.2 and 5.3). We went further.

The concept of privilege mode operation was an obvious area for exploration. When we examined the code from the perspective of privileges and how they were used, we identified another fundamental flaw that we named shedding privileges. By considering changes to the code, we devised another defensive programming practice (Section 4.2) and a design note (Section 5.1). Our analysis was now complete.

We have begun to apply this analysis process to other defects so that we can discover more root causes. We have created a classification of these flaws that we call the root cause taxonomy.[5] This taxonomy will ultimately contain a set of root causes that is gleaned from that detailed analysis and is independent of specific operating systems and programming languages. This forms the basis for the defensive programming practices. We can then add examples for a given programming language and operating system for specific user communities. Through its use, we expect to reduce the instances of the flaws that continue to render systems vulnerable to attack.

[5] Private communication with Jim Ellis.

References

[Braden 89]	*Braden, R. Requirements for Internet Hosts -- Application and Support,* RFC 1123 [online]. Available FTP: <ftp://ftp.isi.edu/in-notes/rfc1123.txt> (1989).
[CERT 96]	CERT/CC. *CERT* Advisory CA-96.21Topic: TCP SYN Flooding and IP Spoofing Attacks* [online]. Available WWW: <http://www.cert.org/advisories/CA-96.21.tcp_syn_flooding.html> (1996).
[CERT 97a]	CERT/CC. *CERT* Advisory CA-97.06 Topic: Vulnerability in rlogin/term* [online]. Available WWW: <http://www.cert.org/advisories/CA-97.06.rlogin-term.html> (1997).
[CERT 97b]	CERT/CC. *CERT* Advisory CA-97.22 Topic: BIND - the Berkeley Internet Name Daemon* [online]. Available WWW: <http://www.cert.org/advisories/CA-97.22.bind.html> (1997).
[Dik 97]	Dik, Casper. *Re: Exploit for Buffer Overflow in /bin/eject - Solaris 2.X* [online]. Available WWW:<http://geek-girl.com/bugtraq/1997_1/0289.html> (1997).
[Harrenstien 85]	Harrenstien, K. *DOD Internet Host Table Specification, RFC 952* [online]. Available FTP: <ftp://ftp.isi.edu/in-notes/rfc952.txt> (1985).
[HP 97]	Hewlett-Packard Company. *setresuid, setresgid - Set Real, Effective, and Saved User and Group IDs* [online]. Available WWW: <http://www.software.hp.com/STK/man/11.00/setresuid_2.html> (1997).
[Kantor 91]	Kantor, B. *BSD Rlogin* [online]. RFC 1282. Available WWW: <http://www.sunsite.auc.dk/RFC/rfc/rfc1282.html> (1991).
[Kettlewell 98]	Kettlewell, R. *Protecting Against Some Buffer-Overrun Attacks* [online]. Available WWW: < http://www.greenend.org.uk/rjk/random-stack.text> (1998).

[Lefty 96] Lefty. *Buffer Overruns, What's the Real Story?* [online]. Available WWW: <http://reality.sgi.com/nate/machines/security/stack.nfo.txt> (1996).

[Mudge 95] Mudge. *How to Write Buffer Overflows* [online]. Available WWW: <http://l0pht.com/advisories/bufero.html> (1995).

[Mudge 96] Mudge. *Compromised – Buffer-Overflows, from Intel to SPARC Version 8* [online]. Available WWW: <http://l0pht.com/advisories/buf.ps> (1996).

[One 96] One, Aleph. *Smashing The Stack For Fun And Profit* [online]. Available WWW: <http://reality.sgi.com/nate/machines/security/P49-14-Aleph-One> (1996).

[POSIX 98] Institute of Electrical and Electronics Engineers. *IEEE Standard Portable Operating System Interface for Computer Environments.* IEEE Std 1003.1-1998. New York, New York: Institute of Electrical and Electronics Engineers (1998).

[Prym 96] Prym. *Finding and Exploiting Programs with Buffer Overflows* [online]. Available WWW: <http://reality.sgi.com/nate/machines/security/buffer.txt> (1996).

[Smith 97] Smith, Nathan P. *Stack Smashing Vulnerabilities in the UNIX Operating System* [online]. Available WWW: <http://reality.sgi.com/nate/machines/security/nate-buffer.ps> (1997).

[Sun 97a] Sun Microsystems, Inc. *in.rlogind, rlogind – Remote Login Server* [online]. Available WWW: <http://docs.sun.com/ab2/@LegacyPageView?toc=SUNWab_40_4:/safedir/space3/pkgs/collections/ab1/SUNWaman/toc/REFMAN1M:0155_in.rlogind.1m;bt=man+Pages(1M)%3A+System+Administration+Commands;ps=ps/SUNWab_40_4/REFMAN1M/0155_in.rlogind.1m> (1996).

[Sun 97b] Sun Microsystems, Inc. *Executable Stacks and Security* [online]. Available WWW: <http://docs.sun.com/ab2/coll.47.4/SYSADMIN1/@Ab2PageView/91913?DwebQuery=user+and+stack#FirstHit> (1997).

[Sun 97c] Sun Microsystems, Inc. *rcmd, rresvport, ruserok - Routines for Returning a Stream to a Remote Command* [online]. Available WWW: <http://docs.sun.com/ab2/@LegacyPageView?toc=SUNWab_40_3:/safedir/space4/pkgs/addoldversion/ab1/SUNWaman/toc/REFMAN3:1020_rcmd.3n;bt=man+Pages(3)%3A+Library+Routines;ps=ps/SUNWab_40_3/REFMAN3/1020_rcmd.3n> (1997).

[Sun 97d] Sun Microsystems, Inc. *setuid, setegid, seteuid, setgid - Set User and Group IDs* [online]. Available WWW: <http://docs.sun.com/ab2/@LegacyPageView?toc=SUNWab_40_3:/safedir/space4/pkgs/addoldversion/ab1/SUNWaman/toc/REFMAN2:0127_setuid.2;bt=man+Pages(2)%3A+System+Calls;ps=ps/SUNWab_40_3/REFMAN2/0127_setuid.2> (1997).

[Sun 97e] Sun Microsystems, Inc. *setreuid - Set Real and Effective User IDs.* [online]. Available WWW: <http://docs.sun.com/ab2/@LegacyPageView?toc=SUNWab_40_4%3A%2Fsafedir%2Fspace3%2Fpkgs%2Fcollections%2Fab1%2FSUNWaman%2Ftoc%2FREFMAN2%3A0146_setreuid.2;bt=man+Pages(2)%3A+System+Calls;ps=ps%2FSUNWab_40_4%2FREFMAN2%2F0146_setreuid.2> (1997).

[Sun 97f] Sun Microsystems, Inc. *setjmp, sigsetjmp, longjmp, siglongjmp - Non-Local goto* [online]. Available WWW: <http://docs.sun.com/ab2/@LegacyPageView?toc=SUNWab_40_4:/safedir/space3/pkgs/collections/ab1/SUNWaman/toc/REFMAN3:2172_setjmp.3c;bt=man+Pages(3)%3A+Library+Routines;ps=ps/SUNWab_40_4/REFMAN3/2172_setjmp.3c> (1997).

[Venema 97] Venema, Wietse. *TCP Wrappers* [online]. Available WWW: <ftp://ftp.win.tue.nl/pub/security/tcp_wrappers_7.6.BLURB> (1997).

REPORT DOCUMENTATION PAGE			*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.			
1. AGENCY USE ONLY (LEAVE BLANK)		2. REPORT DATE November 1998	3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE rlogin (1): The Untold Story			5. FUNDING NUMBERS C — F19628-95-C-0003
6. AUTHOR(S) Lawrence R. Rogers			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-98-TR-017
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/DIB 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-98-017
11. SUPPLEMENTARY NOTES			
12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12.B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS)			
14. SUBJECT TERMS CERT® Coordination Center, coding defects, design notes, defensive programming practice, mitigation strategies, rlogin program, vulnerability analysis			15. NUMBER OF PAGES 40
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

Coding defects account for a significant portion of the reports received by the CERT® Coordination Center. Through in-depth analysis of these reports and generalizing our findings from those analyses, we have begun to create guidelines for mitigation strategies for existing defects and avoidance strategies when coding new software. In this document, we report the results of our analysis of the well-known defect in the rlogin **program. We discuss the coding defect in detail, three mitigation strategies devised to remedy the defect, and two avoidance strategies offered as a guide to reducing the instances of similar coding defects in new programs. We end with three design notes aimed at eliminating these defects at the hardware and protocol design level.**

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102